SOLID: Five basic principles of class design

Classes are the building blocks of applications. Just like bricks in a building. To understand whether a class is properly written, you can check how it measures up to "quality standards". In Java, these are the so-called SOLID principles, and we're going to talk about them

|  |  |
| --- | --- |
| **Principle** | **Description** |
| **Single Responsibility Principle** | Each class should be responsible for a single part or functionality of the system. |
| **Open-Closed Principle** | Software components should be open for extension, but not for modification. |
| **Liskov Substitution Principle** | Objects of a superclass should be replaceable with objects of its subclasses without breaking the system. |
| **Interface Segregation Principle** | No client should be forced to depend on methods that it does not use. |
| **Dependency Inversion Principle** | High-level modules should not depend on low-level modules, both should depend on abstractions. |

1. **Single Responsibility Principle :**

Every class in Java should have a single job to do. To be precise, there should only be one reason to change a class

Such classes will always be easy to modify if necessary, because it is clear what the class is and is not responsible for. In other words, we will be able to make changes and not be afraid of the consequences, i.e. the impact on other objects

**Example :**

|  |
| --- |
| class Book {  String title;  String author;  // other book related attributes and methods  }  class BookPrinter {  void printBook(Book book) {  // print logic  }  } |

1. **Open-Closed Principle:**

Software entities (e.g., classes, modules, functions) should be *open* for an extension, but *closed* for modification.

This means that it should be possible to change a class's external behavior without making changes to class's existing code. According to this principle, classes are designed so that tweaking a class to fit specific conditions simply requires extending it and overriding some functions.

Consider the below method of the class **VehicleCalculations**:

|  |
| --- |
| public class VehicleCalculations {  public double calculateValue(Vehicle v) {  if (v instanceof Car) {  return v.getValue() \* 0.8;  if (v instanceof Bike) {  return v.getValue() \* 0.5;  }  } |

Suppose we now want to add another subclass called **Truck**. We would have to modify the above class by adding another if statement, which goes against the Open-Closed Principle.

A better approach would be for the **subclasses** Car and Truck to override the **calculateValue** method:

**Example :**

|  |
| --- |
| public class Vehicle {  public double calculateValue() {...}  }  public class Car extends Vehicle {  public double calculateValue() {  return this.getValue() \* 0.8;  }  public class Truck extends Vehicle{  public double calculateValue() {  return this.getValue() \* 0.9;  } |

1. **Liskov Substitution Principle :**

This is a variation of the open closed principle that we mentioned earlier. It can be defined as follows: objects can be replaced by objects of subclasses without changing a program's properties.

This means that a class created by extending a base class must override its methods so that the functionality is not compromised from the client's point of view. That is, if a developer extends your class and uses it in an application, he or she should not change the expected behavior of any overridden methods.

**Example :**

|  |
| --- |
| class Bird {  void fly() {  // fly logic  }  }  class Duck extends Bird {  @Override  void fly() {  // fly logic for Duck  }  } |

1. **Interface Segregation Principle :**

This principle states that no client should be forced to depend on interfaces they do not use. Essentially, it suggests that it is better to have many smaller, specific interfaces than one large, general interface.

***Example*:**

Let's say we have a Document interface with methods for **open()**, **save()**, and **print()**. If we have a ReadOnlyDocument class, it doesn't make sense for it to have a **save()** method because it's read-only. This is where ISP comes into play.

|  |
| --- |
| interface Document {  void open();  void save();  void print();  }  interface Openable {  void open();  }  interface Printable {  void print();  }  class ReadOnlyDocument implements Openable, Printable {  public void open() {  // open logic  }  public void print() {  // print logic  }  } |

1. **Dependency Inversion Principle** :

This principle states that high-level modules should not depend on low-level modules. Both should depend on abstractions. Additionally, abstractions should not depend upon details. Details should depend upon abstractions.

In simpler terms, DIP promotes the use of interfaces for the purpose of decoupling, where lower level classes implement interfaces that are used by higher level classes.

**Example:**

Let's consider a **TextEditor** class that depends on a **SpellChecker** class. In a scenario where the **TextEditor** class is a high-level module and the **SpellChecker** class is a low-level module, the high-level module would directly depend on the low-level module, which is not ideal.

|  |
| --- |
| class SpellChecker {  public void checkSpelling() {  // spell checking logic  }  }  class TextEditor {  private SpellChecker checker;  public TextEditor() {  this.checker = new SpellChecker();  }  public void checkDocumentSpelling() {  checker.checkSpelling();  }  } |

By applying DIP, we would introduce an interface, say **ISpellChecker**, that the **SpellChecker** class implements. The **TextEditor** class would then interact with the **ISpellChecker** interface, therefore depending on an abstraction rather than a concrete class.

|  |
| --- |
| interface ISpellChecker {  void checkSpelling();  }  class SpellChecker implements ISpellChecker {  public void checkSpelling() {  // spell checking logic  }  }  class TextEditor {  private ISpellChecker checker;  public TextEditor(ISpellChecker checker) {  this.checker = checker;  }  public void checkDocumentSpelling() {  checker.checkSpelling();  }  } |

In this way, the **TextEditor** class is decoupled from the **SpellChecker** class, and both depend on the **ISpellChecker** abstraction.

**Association**

If two classes in a model need to communicate with each other, there must be a link between them, and that can be represented by an association (connector).

Tables will be Student and Instructor on Db.

A single student can associate with multiple teachers:

Association multiplicity example 1

The example indicates that every Instructor has one or more Students:

Association multiplicity example 2

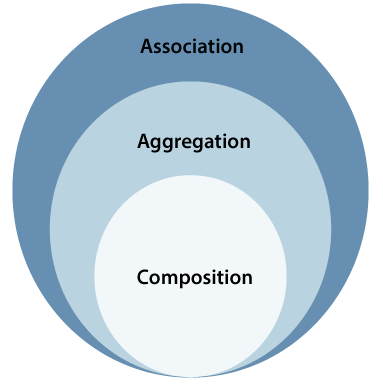
We can also indicate the behavior of an object in an association (i.e., the role of an object) using role names.

Association multiplicity example 3

**Example :**

A tutor can associate with multiple students, or one student can associate with multiple teachers.

The composition and aggregation are two subsets of association. In both of the cases, the object of one class is owned by the object of another class; the only difference is that in composition, the child does not exist independently of its parent, whereas in aggregation, the child is not dependent on its parent i.e., standalone. An aggregation is a special form of association, and composition is the special form of aggregation.

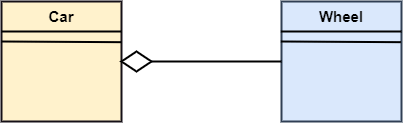


**Aggregation :**

It represents has a relationship. It is more specific than an association. It describes a part-whole or part-of relationship. It is a binary association, i.e., it only involves two classes. It is a kind of relationship in which the child is independent of its parent.

**Example:**

Here we are considering a car and a wheel example. A car cannot move without a wheel. But the wheel can be independently used with the bike, scooter, cycle, or any other vehicle. The wheel object can exist without the car object, which proves to be an aggregation relationship.

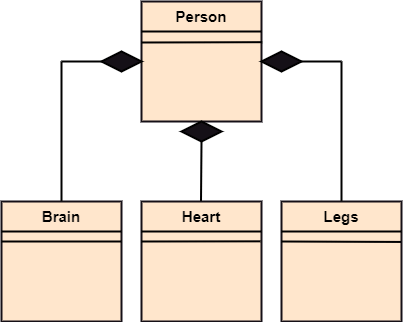


**Composition :**

The composition is a part of aggregation, and it portrays the whole-part relationship. It depicts dependency between a composite (parent) and its parts (children), which means that if the composite is discarded, so will its parts get deleted. It exists between similar objects.

**Example:**

The composition association relationship connects the Person class with Brain class, Heart class, and Legs class. If the person is destroyed, the brain, heart, and legs will also get discarded.



**Design Patterns**

A design patterns are **well-proved solution** for solving the specific problem/task

**Advantage:**

* They are reusable in multiple projects.
* They provide the solutions that help to define the system architecture.
* They capture the software engineering experiences.
* They provide transparency to the design of an application.
* They are well-proved and testified solutions since they have been built upon the knowledge and experience of expert software developers.
* Design patterns don?t guarantee an absolute solution to a problem. They provide clarity to the system architecture and the possibility of building a better system.

**Types:**

1. **Creational Design Pattern**

Creational design patterns are concerned with the way of creating objects. These design patterns are used when a decision must be made at the time of instantiation of a class (i.e. creating an object of a class).

But everyone knows an object is created by using new keyword in java. For example:

StudentRecord s1=**new** StudentRecord();

* Factory Method Pattern
* Abstract Factory Pattern
* Singleton Pattern

etc.

**Singleton Pattern**

Singleton is a creational design pattern that ensures a class has only one instance while providing a global access point to this instance. It's called 'singleton' because it restricts instantiation of a class to a single instance

|  |
| --- |
| public class Singleton {  private static Singleton uniqueInstance;  private Singleton() {}  public static synchronized Singleton getInstance() {  if (uniqueInstance == null) {  uniqueInstance = new Singleton();  }  return uniqueInstance;  }  } |

1. **Structural Design Pattern**

The structural design patterns simplify the structure by identifying the relationships.

These patterns focus on, how the classes inherit from each other and how they are composed from other classes.

* [Composite Pattern](https://www.javatpoint.com/composite-pattern)
* Proxy pattern

Etc.

1. **Behavioral Design Pattern**

the interaction between the objects should be in such a way that they can easily talk to each other and still should be loosely coupled.

That means the implementation and the client should be loosely coupled in order to avoid hard coding and dependencies.

* Iterator Pattern
* Template Pattern
* Null Object

Etc.